(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
14 August 2003 (14.08.2003)

PCT

(10) International Publication Number
WO 03/067427 A2

(51) International Patent Classification[7]: G06F 9/40

(21) International Application Number: PCT/GB03/00518

(22) International Filing Date: 4 February 2003 (04.02.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
02250817.0 7 February 2002 (07.02.2002) EP

(71) Applicant *(for all designated States except US)*: **BRITISH TELECOMMUNICATIONS PUBLIC LIMITED COMPANY** [GB/GB]; 81 NEWGATE STREET, LONDON EC1A 7AJ (GB).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **BUCKLAND, Mark, Adam** [GB/GB]; FLAT 5, ST ANDREWS HOUSE, THE STREET, RUSHMERE ST ANDREW, IPSWICH, Suffolk IP5 1DL (GB). **DUFRENE, Krzysztof, Zbigniew** [PL/PL]; UL.KLAUDYNY 12/71, 01-684 WARSAW (PL).

(74) Agent: **NASH, Roger, William**; BT GROUP LEGAL INTELLECTUAL PROPERTY DEPARTMENT, Holborn Centre, 8th floor, 120 Holborn, London EC1N 2TE (GB).

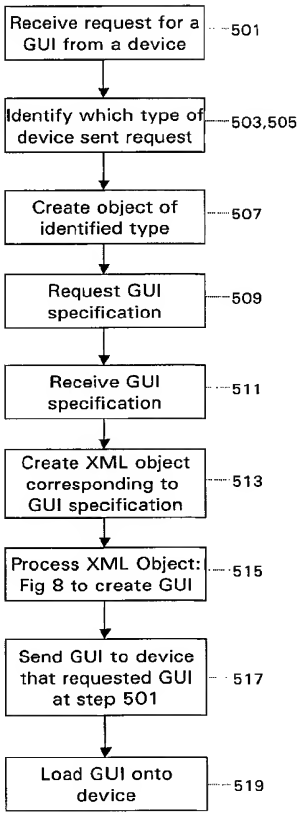
(81) Designated States *(national)*: CA, JP, US.

(84) Designated States *(regional)*: European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, SE, SI, SK, TR).

Published:
— *without international search report and to be republished upon receipt of that report*

[Continued on next page]

(54) Title: GRAPHICAL USER INTERFACE



(57) Abstract: The present invention is concerned with rapidly changing requirements for graphical user interfaces. The range of mobile devices and the rate of change in content of information accessible therefrom are increasing. One of the drawbacks of having such a range of devices is that there is a corresponding range of configurations and capabilities, and, in general, each device requires a different Graphical User Interface (GUI). For companies developing products for such a range of devices, the infrastructure and API requirements can be highly disparate and the development time can be significant. To alleviate these problems, an embodiment of the invention provides a means of specifying the requirements of a GUI in a single specification, which includes generic descriptions of GUI elements and can be interpreted in respect of any type of device. The embodiment has access to device-specific GUI items, so that, once the device type and a generic GUI element has been identified, the device specific GUI item corresponding thereto is readily identifiable. Preferably the specification is written as an XML file, which means that it can be stored on a centralised server, and is thus accessible by well-known web server access techniques. In one embodiment the specification is processed, and GUI created, from the server computer 301, which then sends the created GUI to an appropriate device for display thereon.

WO 03/067427 A2

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

GRAPHICAL USER INTERFACE

The present invention relates to graphical user interfaces, and is suitable particularly, but not exclusively, for network-connected devices.

The demand for network-connected devices is growing at a significant rate. In response to this demand, new classes of devices such as smart cellular telephones, pagers, Personal Digital Assistants (PDA, which includes any small mobile hand-held device that provides computing and information storage and retrieval capabilities for personal or business use) etc. are continually being developed. In addition, traditional consumer electronics such as televisions, Digital Versatile Disc and Compact Disc (DVD and CD) players are equipped with new capabilities.

In order to develop applications on any one of these ever increasing device types, Sun™ Microsystems has developed a new Java™ application environment, called "Java 2 Platform, Micro Edition Solution" (J2ME™), which comprises bespoke configurations and profiles for different lightweight device types. Among other things, these configurations and profiles can be used to create a Graphical User Interface (GUI) on a device.

A configuration is comprised of a virtual machine, which, as is known by those skilled in the art, enables Java™ applications to run on the device (for further details of Java virtual machine, and how it interoperates with a microprocessor running on a device, the reader is referred to "The Java™ Virtual machine specification", Sun Microsystems Chapter 1.2, Lindholm, T., Yellin, F. 1999. In addition a configuration is comprised of core libraries, classes and application programming interfaces (APIs). There are currently (January 2002) two configurations: the co-called Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC). The former configuration type, CLDC, is suitable for devices with constrained processing capabilities and memory resources, while the latter configuration type, CDC, is suitable for devices with more resources.

A profile is a specification that defines device specific application programming interfaces (APIs), such as device display features. An example of the relationship between configurations and profiles is shown in Figure 1, which shows two types of profile types: MIDP 101 and Personal 103. MIDP 101 is a profile type that is suitable

for wireless and mobile devices, and links into the CLDC 105 configuration, while the Personal profile 103, together with the Foundation profile 106, is suitable for devices with more computational power, and co-operates with the CDC 107 configuration. As can be seen from Figure 1, each configuration CLDC 105, CDC 107 interoperates with a Java virtual machine KVM 109, CVM 111 respectively.

Clearly with this approach, Java™ technology is moving away from the "Write Once, Run Anywhere" paradigm, and developers are once again forced to write different applications for different platforms.

According to a first aspect of the present invention there is provided apparatus for creating a Graphical User Interface for a device, comprising

a store arranged to store a plurality of processable procedures, each being associated with one or more device types;

receiving means arranged to receive a device independent Graphical User Interface specification and data identifying the device;

processing means arranged to process the received Graphical User Interface specification in accordance with the received identifying data, so as to identify, from the store, a plurality of procedures corresponding to the identified device; and

combining means arranged to combine the identified procedures into a program for processing by the device, thereby creating the Graphical User Interface.

Conveniently the device-independent graphical user interface specification identifies display requirements of a graphical user interface. Embodiments of the invention identify a device type, and, depending on the device type, interpret the specification in accordance with device-dependent display capabilities. These device-dependent capabilities can be derived from software that has been written for a particular device.

Advantageously at least some of the plurality of procedures are written in accordance with object-oriented programming, so that at least some procedures have, associated therewith, a class, which is associated with a device type. The apparatus further includes means arranged to instantiate an object of the identified device type and to pass the instantiated object to the processing means, the

processing means thereby having access to the procedures associated with the instantiated object.

The processing means is then arranged to receive a reference to procedures corresponding to items in the received Graphical User Interface specification

Preferably the GUI specification is written in the Extensible Mark-up Language (XML). Alternatively the GUI specification could be written in User Interface Markup Language (UIML), which is an XML language specifically for defining user interfaces. As a further alternative the GUI specification could be written using a proprietary descriptive language having a bespoke standard for the encoding of items such as GUI widgets, symbols and layouts etc.

Conveniently, for each item in the GUI specification, embodiments of the invention have access to a repository of functions for each device type, so that if, for example, the XML specification includes the text<display>, and the device type is identified as being a wireless device, an object corresponding to a wireless device is created and a function of that object, which has the capability of creating a display on a wireless device, is then invoked.

Preferably the processable procedures include procedures for processing one or more events in accordance with user input received via the application program. Conveniently the apparatus includes means to select one or more of the said procedures for processing one or more events on the device in accordance with items in the Graphical User Interface specification.

According to the invention there is also provided a method corresponding to the apparatus described herein.

The apparatus can either be run on a server apparatus, in which case the Graphical User Interface, once created, is transmitted to a device for which it is intended. Alternatively the apparatus can be run directly on the device itself.

The library functions provided by "Abstract Factory™" can be used to invoke display components as a function of device type (described by Gamma, E., Helm, R., Johnson, R., and Vlissides, J. in "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Reading, MA, at pp. 87 – 95). These library functions are written in accordance with the object-oriented programming paradigm, and provide polymorphism of objects. Polymorphism is a concept whereby the

functionality of an object is defined by its run-time type. Polymorphism is implemented via an interface comprising one base class that contains all methods common to different behaviours. The common methods are defined, in the base class, as being *virtual*. The different behaviour is provided by different child classes (derived from the base class, also known as a sub-class), which implement the methods that were defined as virtual in the base class.

An array can then be created, whose elements are declared as instances of the base class; at run-time (i.e. when the elements are instantiated), any sub-class of the known base class can replace an element of the array, thereby dynamically defining the behaviour of the object.

This functionality can be described by an example: consider a program that draws shapes on the screen. First, we declare a base class, *Figure*, having one or more virtual methods therein, which are declared as "abstract". In this example, base class *Figure* includes virtual method *Draw*.

Abstract methods exist only to provide a gateway to multiple different forms of a method, and are thus designed to be overridden by a child class. As a result the *Figure* class (base class) does not include an implementation of the *Draw* method (In C++, the *Draw* method is called **pure virtual member function**, in Java the Draw method is simply called an abstract method).

Continuing with the example, next we define child classes: Square and Triangle, which implement the method *Draw*:

```
// Common for Square and Triangle.
Polygon = class(Figure)
private
  a: integer; // Length of the side.
end;

Square = class(Polygon)
public
  procedure Draw; override; // Overrides Figure.Draw.
end;

Triangle = class(Polygon)
public
  procedure Draw; override; // Overrides Figure.Draw.
end;

// Include some implementation for Square.Draw, Triangle.Draw (not shown)
```

Next we create a data structure, *Figures* array:

```
Figures = array[0..2] of Figure;
```

which is actually an array of pointers declared to point to objects of *Figure* class. This means that, at run-time, we can place children of the *Figure* class in the array. i.e.:

```
var
  F: Figures;
  i: integer;

F[0] = Square.Create;
F[1] = Triangle.Create;

// Draw figures.
//We don't need to know the actual class of these objects!
for i:=Low(F) to High(F) do
  F[i].Draw;
```

The Abstract Factory™ libraries include abstract base classes and child classes that provide polymorphism. Referring to Figure 2, instead of hard-coding display widgets into an application A (e.g. an application called "GUI creation"), there is an interface 201, which is representative of a virtual method within a base class, and which can be passed as a parameter to the Application A. When application A is run, the interface 201 is cast to whichever object corresponds to the client type, and sent as a parameter to the application A. Thus, for example if a client C is running Motif, then when the Application A requests 212 a scrollbar, the interface 201 will automatically pass, to the Application A, a reference 214 to a MotifScrollBar object.

An Open Source product called "Abstract User Interface Toolkit" (AUT) also makes use of the Abstract Factory design approach. GUI requests are written in an XML specification, and the client translates the specification in order to identify the object to be requested. Display components corresponding to the identified object are passed to the display application via the Interface 201 as described above. This Open Source product has been written for "real" windows (in this case Motif, Macintosh OS X and Win32) that run on a single device (conventional PCs). Each window type relates to a different operating system (Unix, Mac and Windows respectively) and has a separate XML specification corresponding thereto. By comparison, in embodiments of the invention a single GUI specification is written and this can be used for any device type.

Essentially the "Abstract User Interface Toolkit" is designed for conventional desktop machines, where the actual windowing functionality is identical, but the means of invoking the functionality is not. By comparison, embodiments of the

invention are concerned with devices where the graphical interface functionality varies considerably between device types: cell phones and pagers, for example, have small screens, limited battery life and low memory resources; whereas entertainment environments offer high speed connectivity to a network and have around 16 MB memory.

Further features and advantages of the invention will be apparent from the following description of preferred embodiments of the invention, which are given by way of example only and with reference to the accompanying drawings, in which

Figure 1 shows an overview of the "Java 2 Platform, Micro Edition Solution" (J2ME™) application environment with which embodiments of the invention interoperate;

Figure 2 is a schematic representation of "Abstract Factory™" design utilised by embodiments of the invention;

Figure 3 is a schematic representation of the environment in which embodiments of the invention operate;

Figure 4 is a schematic representation of an embodiment of the invention generally referred to as GUI engine;

Figure 5 is a flow diagram showing steps involved in a first embodiment of the invention;

Figure 6 is a schematic representation of an object created by the parsing program forming part of the GUI engine shown in Figure 4;

Figure 7 is a schematic representation of the functionality of the interface forming part of the GUI engine of Figure 4;

Figure 8 is a flow diagram showing steps involved in processing the object shown in Figure 6;

Figures 9a and 9b are illustrations of a GUI created by the GUI engine shown in Figure 4; and

Figure 10 is a schematic representation of processing of an event listening program.

OVERVIEW OF AN EMBODIMENT OF THE INVENTION

Referring to Figure 3, an embodiment of the invention runs on a server computer 301, which is in communication with a range of devices 303a 303b, 303c

via a public land mobile network (PLMN) (e.g. a GSM - compatible digital cellular network) N1 or, if the devices 303a, 303b, 303c are equipped with Wireless Ethernet network (WLAN) technology, via a 802.11-compatible Wireless Ethernet network (WLAN) N2 (or any other short range network). Additionally, or alternatively the devices 303a, 303b, 303c could communicate with the server 301 via GPRS and/or UMTS, which are mobile networks capable of sending and receiving packet switched data.

The PLMN network N1 can be connected via a gateway G1 to a public switched network (PSTN) N3. Figure 3 shows 3 device types only, but many more are possible.

The PSTN N3 is interconnected with an integrated services digital network (ISDN) N4 via a gateway G2 (e.g. a local or international switching centre), and is connected via an ISDN line L1 to the server computer 301, and to local area network (LAN) N5. One or more database servers 01 are accessible by the server computer 301 via the local area network N5. In addition the ISDN network N4 is connected to the WWW via line L2.

Information sources, held for example on servers IS1-IS4, are distributed throughout the networks N1-N6. Only four servers are shown, however other distributions are envisaged. A user operating a smart phone device 303a may wish to receive information from any or all the information sources in the system.

With the proliferation and continuing development of mobile devices 303a – 303c, and the rate of change to the type and content of information accessible therefrom, it is vital that the user interface can be easily modified. However, one of the drawbacks of having such a range of devices is that there is a corresponding range of configurations and capabilities, meaning that each device requires a different GUI. For companies developing products for such a range of devices, the infrastructure and API requirements can be highly disparate and the development time can be significant.

To alleviate these problems, an embodiment of the invention provides a means of specifying the requirements of a GUI in a single specification, which includes generic descriptions of GUI elements and can be interpreted in respect of any type of device. The embodiment has access to device-specific GUI items, so that, once the device type and a generic GUI element has been identified, the device specific GUI

item corresponding thereto is readily identifiable. Preferably the specification is written as an XML file, which means that it can be stored on a centralised server, and is thus accessible by well-known web server access techniques. In one embodiment the specification is processed, and GUI created, from the server computer 301, which then sends the created GUI to an appropriate device for display thereon.

A particular feature of the invention is that, because a single GUI specification is used, the same information and menu options will be available on all devices used by a user (albeit arranged differently). It is well recognised that humans prefer to use devices with which they feel familiar, because it means that information is delivered and/or accessed to and/or by a user in a manner with which they are accustomed. Thus this conformance of GUI components across devices is thus a significant advantage.

Referring now to Figure 4, the server computer 301 comprises a central processing unit (CPU) 401, a memory unit 403, an input/output device 405 for connecting the server computer to the devices 303a, 303b, 303c and to the database 01; storage 407, and a suite of operating system programs 409, which control and co-ordinate low level operation of the server computer 301.

Generally an embodiment of the invention is referred to as a GUI engine 400, and comprises at least some of programs 411, 413, 415, 417. These programs, which are described in more detail below, are stored on storage 407 and are processable by the CPU 401. The GUI engine 400 also includes, or has access to, a database 01, which stores methods corresponding to device types. It is understood that each of the programs 411, 413, 415, 417 could comprise a suite a computer programs, and is preferably written in an object-oriented programming language, such as Java™.

The programs include a program 411 for detecting a type of device for which the GUI is intended, a program 413 for requesting and/or receiving a specification of the GUI requirements, a program 415 for parsing the specification, and a program 417 for processing the parsed specification so as to create a GUI comprising features that are appropriate to the detected device type. Conveniently at least some

of the programs 411, 413, 415, 417 make use of, and extend, library functions supplied by Abstract Factory™ described above.

The specification of GUI requirements can be written using the Extensible Mark-up Language (XML). This specification is device-independent, so that, for a particular GUI format, a single specification applies for all of the devices 303a, 303b, 303c. Essentially, once the detecting program 411 has detected a device type, it creates an object of that device type. When the processing program 417 processes the specification, the methods and features of the created object are automatically run and retrieved, thereby creating a GUI that has been customised for the detected device type.

Referring to Figures 5, 6, 7 and 8, an embodiment of the invention will now be described in more detail. Figure 5 is a flow diagram showing steps carried out by the GUI engine 400 when creating a GUI for a smart phone 303a, and Figure 6 is a schematic representation of a device-independent GUI specification. Figure 7 is a schematic block diagram showing device-dependent objects and methods thereof that are stored in the database 01 and accessed by the processing program 417. In Figure 7 the complete lines with arrows indicate process steps, and the dash-dotted lines indicate access relationships. Figure 8 is a flow diagram showing steps carried out by the processing program 417.

In the description of the current embodiment, it is assumed that the programs 411, 413, 415, 417 operate in "object-oriented" fashion; that is to say that the data associated with the programs is "encapsulated" so as to be accessible only by associated control programs, acting in response to "messages" (which need not, however, be physically transmitted but could simply be data passed via the stack of a single computer).

Referring first to Figure 5, the GUI engine 400 receives 501 a request for a GUI from the smart phone 303a, and sends this request to the detecting program 411, which is activated to identify the type of device that has sent the request. When the GUIs are created using Java™, they are run on a Java virtual machine running on the device 303a. The type of Java virtual machine will vary from device to device, depending on the configuration 105, 107 running thereon, so identifying the type of Java virtual machine provides a means of detecting the type of device.

Accordingly, at step 503 the detecting program 411 sends a request for the type of virtual machine running on the device 303a. This involves remotely invoking a system method, getProperty () on the device, via Remote Method Invocation (RMI). RMI is part of the Java™ programming language library functions and involves a local surrogate (stub) object (not shown) on the server machine 301 managing invocation on a remote object (device 303a). The system method getProperty () is available from the configuration API running on the requesting device 303a. The following are examples of the request messages invoked on the device at step 503:

J2SE

To check if the device is running J2SE the detecting program 411 asks if the system property `java.runtime.name` contains the value `Java(TM) 2 Runtime Environment`. If the string is anything else or `null` then the platform is not J2SE. Examples of devices that run J2SE are laptops and desktop PCs.

```
e.g.
public static final boolean isJ2SE;

String property = System.getProperty("java.runtime.name");
if (property != null)
  isJ2SE = (property.indexOf("Java(TM) 2 Runtime Environment") != -1);
else
  isJ2SE = false;
```

Personal Java

To check if the device is running pJava the detecting program 411 asks if the system property `os.name` contains the value `Windows CE`. As Windows CE only runs Personal Java the platform will be pJava (pJava devices include the iPAQ (high-spec PDA)). If the string is anything else or `null` then the platform is not pJava.

```
e.g.
public static final boolean isPjava;

property = System.getProperty("os.name");
if (property != null)
  isPjava = (property.indexOf("Windows CE") != -1);
else
  isPjava = false;
```

It should be noted that during 2002 Sun™ Microsystems will replace PersonalJava™ with the Configuration/Profile architecture, (CDC/Foundation Profile/Personal Profile) shown in Figure 1. As a result the above code will be modified so that, when sent a getProperty() request, the System returns an identifier indicative of a device

operating in accordance with a Config/Profile. The skilled person would readily appreciate how to make such a modification.

MIDP

To check if the device is running MIDP the detecting program 411 asks if the `microedition.profiles` system property contains the string `MIDP`. If the string is anything other than MIDP or null then the platform is not MIDP. Examples of devices that run MIDP are smart phones and Palm Pilots.

```
e.g.
public static final boolean isMIDP;

property = System.getProperty("microedition.profiles");
  if( profiles != null )
    isMIDP = ( property.indexOf("MIDP-") != -1);
  else
    isMIDP = false;
```

By invoking a series of `getProperty()` messages the detecting program 411 identifies 505 which platform, and thus which type of device, has sent the request message.

The detecting program 411 then creates 507 an object OD (not shown) of this identified device type (equivalent to `square.create` in the example given in the introduction section). Each device object has methods and variables corresponding thereto, so that, by creating an object of the detected device type, these associated methods and variables can thereafter be invoked.

The GUI engine 400 then activates the receiving program 413 to retrieve a specification of the GUI requirements. When the specification is written in XML, it can be accessed from a web server, via a Uniform Resource Locator (URL) submitted with the request sent at step 501. If the request (step 501) does not include a URL, or some indication of an access point for the specification, the receiving program 413 can access a default GUI specification (or a default URL) from the GUI engine 400. Accordingly, assuming a URL is submitted together with the request at step 501, the receiving program 413 sends 509 a request for an XML file to a web server at the submitted URL, and receives 511 an XML specification therefrom.

The parsing program 415 is then activated to parse the XML specification. In one arrangement the parsing program 415 creates and stores 513 the specification as an XML object 600 comprising a series of nodes, each of which corresponds to a

tag in the XML specification, and is readily processable by the processing program 417. An overview of XML and tags is given on pages 2 – 5 of McLaughlin, B., *Java and XML*, O'Rielly, 2000. (ISBN 0-596-00016-2), and an overview of parsing of XML is given on page 23, and 46-57.

An example of such an XML specification is given below (where a tag is given by :<tag>), and a parsed XML object corresponding thereto is schematically shown in Figure 6:

```
<display>                                  601    // display tag:"display" root element
        <panel>                            603    // panel tag: "external" panel
            <position></position>                        605
            <name>3rd form</name>                        607
            <layout>                                     609
                <name>BorderLayout</name>                611
            </layout>
            <panel>                        615           // inner panel
                <position>CENTER</position>              617
                <name>1st form</name>                    619
                <layout> </layout>                       621
            </panel>
           <choicegroup>                                 633
                 <position></position>                   635
                 <name>This is Choice Group</name>       637
             <type>MULTIPLE</type>                       639
             <choice>A</choice>                          641
                         <choice>B</choice>              643
                         <choice>C</choice>              645
           </choicegroup>
           …
</display>
```

The XML object 600 created at step 513 is then processed, step 515, by the processing program 417. This involves processing each node of the XML object 600, as described in detail with respect to Figure 8 below.

As stated above, at least some of the programs 411 – 417 make use of the Abstract Factory™ libraries (described above with reference to Figure 2). Referring to Figure 7, the GUI engine 400 has an interface 701, which is arranged to receive a device-independent request for some aspect of a GUI from the processing program 417, and, depending on the object OD created at step 507, the interface 701 will refer to a device-dependent method relating to the request. These methods, which are described in more detail below, can be stored in the database 01 and are automatically accessible by the object when it is created at step 507.

Figure 7 illustrates the functionality of the interface 701 for the first node <display> of the example XML specification given above. Referring to Figure 7 the processing program 417 sends S 7.1 a request for a "display" to interface 701. The interface 701 returns a reference to a display function S 7.2 of the object created at

step 507 back to the processing program 417 (essentially OD.createDisplay() (cf `F[i].draw = square.create.draw()` in the example given in the introduction)). This means that, at runtime, the specific implementation of the createDisplay() method corresponding to object OD will be invoked.

Prior to sending the request at step S 7.1, the processing program 417 has to identify which generic GUI element to request. For example, this can involve accessing a Hashtable data structure, which stores relationships between XML tags and generic GUI elements. A generic GUI element corresponding to an XML tag <name> can be identified by invoking a method to retrieve a node corresponding to <name> from the Hashtable, in accordance, for example, with the following psuedo-code:

```
Hashtable componentMapping;
Go to first node of XML
GenericComponent gc;
Do(whilst not at end of XML structure)
   gc= (GenericComponent) componentMapping.get(elementName);
   gc.create();
   get next element of node
end do
```

Alternatively this can involve processing a sequence of "if ... then" statements, where the "if" is conditional on the type of node (in the code fragment set out below, the type of node is object elementName):

```
if(elementName.equals("display"))
        {
            // create display...
            factory.createDisplay(app);

            number = elementRoot.getChildCount();
           // System.out.println("\n"+number);
            if(number==0) return;
            for(int i=0;i<number;i++)
            {
                if(elementRoot.getType(i)==Xml.ELEMENT)
                {
                    element = elementRoot.getElement(i);
                    processGUISpec(element,null);
                }
            }
        }
```

This alternative example includes several object instantiations equivalent to `Square.create()` that was discussed in the introduction.

Steps S 7.1 and S 7.2 are repeated for each node in a depth first with back tracking manner (i.e. top to bottom, left to right - that is, the first route through the XML object 600 is the left-most nodes, and when a leaf node is encountered, the route is back tracked until an adjacent right hand node is encountered whereupon the route burrows down this adjacent node until a next leaf node is encountered. This process is repeated for all nodes in the XML object 600). Tree structures (such as is shown in Figure 6) and depth first traversal are described in Budd, T. 2001, *Classic Data Structures in Java*, Addison Wesley Longman, Oregon, at, respectively pp 325-330, and pp 343-344.

This method is illustrated in Figure 8, for the example XML specification shown in Figure 6. The processing program 417 reads in 801 the first node in the XML object 600 and checks 803 whether the node is of "display" type. If it is (as it is in the current example), the processing program 417 sends S 7.1 a request for a "display" (via the interface method `createDisplay()`) to object OD (which sub-classes interface 701) created at step 507, and returns S 7.2 the reference back to the processing program 417, which stores 805 this as a reference to a new display area. This means that, at runtime, the specific implementation of the createDisplay() method corresponding to object OD will be invoked.

The processing program 417 then reads in 807 the left-most node 603 and repeats step 803 in respect thereof, moving onto step 809 because the next node is not a "display" node. At step 809 the processing program 417 checks whether the node is of "panel" type. As in this example it is, the processing program 417 sends S 7.1 a request for a "panel" (via the interface method `createPanel()`) to object OD created at step 507, and returns S 7.2 the reference back to the processing program 417. This means that, at runtime, the specific implementation of the createPanel() method corresponding to object OD will be invoked.

The processing program 417 then adds 811 the reference to the panel to the reference stored at step 805. The processing program 417 then checks 813 whether this panel is the first panel node encountered in the XML object 600, and if it is, it sets 815 this panel as the root display element. If this panel is not the first panel node, step 815 is omitted, and the processing program 417 skips to step 817, where the processing program 417 reads in 817 the left-most node 605 and repeats steps 803, 809 in respect thereof.

In the current example this node is neither a "display" nor a "panel" node, so the processing program 417 omits steps 803, 809, and extracts 819 whatever information pertains to the node (for node 605 there is no additional information to be extracted).

Steps 819 – 823 are best illustrated for the case of node 633. When the processing program 417 reaches node 633, it sends a request for a "choice group" (via the interface method `createChoiceGroup()`) to object OD created at step 507, and returns S 7.2 the reference back to the processing program 417. The processing program 417 then adds 823 the reference to the panel set as root display element at step 815.

Whenever the processing program 417 comes across a terminal branch node (e.g. nodes 605, 607a, 609a) it backtracks 880 through the XML object 600 to identify a next untested node (e.g. having reached node 605, the processing program 417 backtracks to node 603).

Returning to Figure 5, once the processing program 417 has processed all of the nodes in the XML object 600, the processing program 417 can be said to have created a bespoke GUI class, and, at step 517, the class is sent to the requesting device (here smart phone 303a) as byte code. The client C, which is running on the device 303a and is in operative association with a Java Virtual Machine, then reads in the byte-code and converts it into a class. Thereafter the client C instantiates a new instance of the class – namely a GUI object – essentially displaying 519 the GUI on the device 303a. This effects "run-time" instantiation of the referenced methods, as described with reference to the *Draw* and *Figure* example in the introduction section.

Figures 9a and 9b each show a GUI that has been created from the same XML specification, for a smart phone and a PDA respectively. The two interfaces show identical information: referring to Figure 9a menu options XmlInfo 901, Alert 903, TxtBox 905 TextArea 907 correspond to XmlInfo 911, Alert 913, TxtBox 915 TextBox 917 on Figure 9b.

Second Embodiment

A second embodiment, generally similar to that described in Figures 1 – 9, will now be described, in which like parts have like reference numerals and will not be

described further in detail. In the first embodiment, one or more server computers 301 receive requests for GUIs from devices. In a second embodiment, each device 303a, 303b, 303c has a GUI engine 400 running thereon, so that a request for a GUI is processed locally on the device itself.

Referring to Figure 10, in order for the device 303a to process user input received via the GUI loaded at step 519, it needs to run event listening code 1001. As is known in the art, event listening programs listen for inputs (e.g. user input via a keyboard, or via a keypad) and have procedures that respond, with predefined actions, to received inputs.

Typically a developer will write event listening code 1001, which corresponds to items in the GUI specification; for each GUI specification, the developer only needs to write one version of event-listening code, which can thereafter be sent to any device.

The event listening code 1001 can be stored on the server computer 301 (e.g. as a library class). When the event listening code 1001 is written in the Java™ programming language, it is compiled, on the server computer 301, into byte-code, sent to the device 303a, and received by a program 1003 running on the device 303a. As stated above, the client C running on the device 303a is in operative association with a Java Virtual Machine, which is arranged to interpret the byte-code of the program 1001 (once translated it into a class file).

The following code fragment is an example of event listening code for handling menu selection (in the present embodiment the event-listening code 1001 is a class that is written using the Abstract Factory™ libraries):

```
// create menulistener for application's menu

menulistener = factory.menuListener(new LEAPAction()
{
    public void action(){
        try{
//get the menu option selected by the user
            int i = factory.getMenuCommand();
            switch(i)
          {
          case 0:
              //when 'XmlInfo' menuitem is pressed, info defined in xml
              //becomes visible
              System.out.println("XmlInfo pressed");
              factory.setDisplay(components.get("XmlInfo"));
              break;
          case 1:
```

```
                //when 'Alert' menuitem is pressed, Alert becomes
                // visible
                System.out.println("Alert pressed");
                factory.setDisplay(components.get("XmlAlert"));
                break;
            case 2:
                //when 'TxtBox' menuitem is pressed TxtBox
                //is made visible on panel4...
                System.out.println("Info pressed");
                factory.addToPanel(components.get("4th form"),
                            factory.textArea("TextBox...","Hello!!!",100),"NORTH");
                break;
            case 3:
                //when 'Panels' menuitem is pressed panel3
                //is made visible...
                System.out.println("Panels pressed");
                factory.setDisplay(components.get("3rd form"));
                break;
            case 4:
                //when 'Help' menuitem is pressed info box
                //is made visible...
                factory.setDisplay(factory.infoScreen("Sorry, no help!!!"));
                System.out.println("Help pressed");
                break;
             default:
                break;
        }
    }catch(Exception e){
          e.printStackTrace();
    }
```

The client C firstly instantiates an object of the device (it can reuse the device object DO created at step 507). Once the JVM has interpreted the received byte-code into an event listening class, the client C instantiates the said class, thereby realising references to certain event listening methods (in the example above, menuListener() and getMenuCommand()) via the interface 701. This essentially involves requesting a reference of a menuListener, or getMenuCommand etc. function of the object DO (i.e. calling "OD.menuListener").

The client C can then invoke the method(s) using the reference(s) that is/are received from the interface 701 at run-time. Considering the example of a pJava device, a reference to the method menuListener() for this device type effectively activates method `ActionListener()` at run-time.

The reference to `menuListener()` is then coupled to components of the GUI, thereby "attaching" the event listeners to the GUI (referring to the code fragment below):

```
// add menulistener to the application's menu

try{
    factory.setMenuListener(components.get("Menu"),menulistener);

}catch(Exception e){
```

```
        e.printStackTrace();
    }
    // components.get("Menu") identifies which GUI component to attach the listener
    // to (here a menu component), and menuListener is effectively a reference to
    // "OD.menuListener()"
```

Mapping between generic GUI elements and device specific GUI items

As stated above, an interface 701 essentially provides a means of implementing mappings between generic GUI elements and device specific GUI items (returning to Figure 7, "Display" is a description of a generic GUI element, and "MIDlet display" is a GUI display item specific to the smart phone 303a). In the first and second embodiments the actual implementation of the mappings is via polymorphism, as each type of device is represented by a sub-class, each of which conform to methods corresponding to their base class, interface 701. In another embodiment the mappings are implemented as a look-up table between generic description and a specific item; this is described in more detail later.

Returning to the first and second embodiments, each specific GUI item involves one or more classes and methods. As will be appreciated, identifying GUI items for some generic GUI elements is non-trivial, not least because the GUI architecture for different Java™ platforms can vary considerably in terms of the relationships between GUI items, the hierarchy of GUI items and the inadequate, and in some instances excessive, equivalencies of GUI objects between platforms.

The following table presents examples of GUI items for MIDP and pJava devices corresponding to generic GUI descriptions. At least some of the items involve bespoke classes and methods (indicated by "bespoke"), while others are provided by the API corresponding to the device (indicated by "API class"). This list is not exhaustive.

Description of Generic GUI element	**MIDP GUI item**	**PJava GUI item**	**Comments**
Display (main window)	Display (API class)	Frame (API class)	

Panel (screen main within window)	Form (API class + bespoke tweaking of methods therein)	Panel (API class)	Panel components can be added using an appropriate layout whereas Form has no layout – it puts all components under one another. In addition Panel can contain other Panels whereas Form cannot contain other forms. These difficulties were solved with a restriction on buttons.
Textfield (allows user to enter text)	Textfield (API class)	Textfield (API class)	
TextArea	Textfield (API class)	TextArea (API class)	TextArea in pJava has no equivalence in MIDP, so TextField can be used. As an alternative, the TextBox class could be used (although TextBox is a separate Screen class, rather than an Item class so it cannot be added to the Form). See parts 907, 917 on Figures 9a, 9b
Date	DateField (API class)	MyDateField (bespoke)	pJava classes do not include date and time fields, so MyDateField was created based on Abstract Windows Toolkit™ (AWT) classes
Time	TimeField (API class)	MyTimeField (bespoke)	pJava classes do not include date and time fields, so MyDateField was created based on AWT classes
ChoiceGroup (allows user to select from some options)	ChoiceGroup (API class)	CheckBox (API class)	
AlertScreen (displays data on the screen)	Alert (API class)	LEAPAlert (bespoke)	LEAPAlert class extends the Dialog class available from the pJava API
Menu (allows user to select from options)	List (API class)	Toolbar & buttons (bespoke)	The menu item in pJava is implemented as a toolbar, which is placed at the top of the frame and includes various buttons
Button (device-specific soft buttons)	Command (API class)	Button (API class)	

Listen (listen for events)	CommandListener (API class)	ActionListener (API class)	

Whenever new XML tags are added, or a generic GUI element corresponding to an XML tag changes, the hashtable (or series of switch statements) described above requires updating, so that, when the graphical user interface specification is processed, the processing program 417 can identify which request to send at step S 7.1.

In a third embodiment, polymorphism is not used. This means that the classes and methods are explicitly written for each device type, and an object of each type of device has to be explicitly instantiated when creating the GUI (i.e. no instantiation of object at run-time). Accordingly at some point prior to step 801, the processing program 417 receives an identifier representative of the said device type and, when processing the XML object 600, it directly invokes classes and methods relating thereto so as to add a specific component to a specific display that relates to that specific platform of the device type. The following pseudo-java code illustrates how this embodiment could operate:

```
PjavaDisplayObject pjavaDisplay; //this object only runs in a pjava environment
MIDPdisplayObject midpDisplay; //this object only runs in a MIDP environment

If (the system is running on pJava)
   Create a gui based on pjavaDisplay;
   processXMLforPjava(pJavaDisplay);
Elseif (the system is running MIDP)
    Create a gui based on midpDisplay;
    ProcessXMLforMIDP(MIDPDisplay);
Endif
```

where `ProcessXMLforPjava(pJavaDisplay)` is defined as:

```
void ProcessXMLforPjava(PjavaDisplayObject  pJavaDisplay){

    Get the GUI element description from the next node;
    While (more nodes exist)

       If (node is x)
           Create a pJava GUI element relating to x and add it to the
pJavaDisplay;
       Elseif (node is y)
           Create a pJava GUI element relating to y and add it to the
pJavaDisplay;
       Elseif (node is z)
```

```
            ...
        endif

    endwhile
}
```

and ProcessXMLforMIDP(MIDPDisplay) is defined as:

```
void ProcessXMLforMIDP(MIDPDisplayObject  midpDisplay){

    Get the GUI element description from the next node;
    While (more nodes exist)
       If (node is x)
           Create a MIDP GUI element of relating to x and add it to the
MIDPDisplay;
       Elseif (node is y)
           Create a MIDP GUI element of relating y and add it to the
MIDPDisplay;
       Elseif (node is z)
           ...
       endif
    endwhile
}
```

A fourth embodiment similarly does not use polymorphism (interface 701). Instead, the mapping between generic GUI description and specific GUI item is provided by a simple lookup table of the following form:

Generic GUI description (node)	Specific GUI item (pJava)	Specific GUI item (MIDP)
Display	Display: invoke method CreateDisplay {...}	Form: invoke method CreateForm {...}

The processing program 417 then operates as follows:

```
void ProcessXMLforMIDP(MIDPDisplayObject  midpDisplay){

    Get the GUI element description from the next node;
    While (more nodes exist)
       Select and create a MIDPGUIItem corresponding to this node from the
mapping
    endwhile
}

void ProcessXMLforpJava(pJavaDisplayObject  pJavaDisplay){

    Get the GUI element description from the next node;
    While (more nodes exist)
       Select and create a pJavaGUIItem corresponding to this node from the
mapping
```

```
    endwhile
}
```

Additional Implementation details:

The programs 411 – 417 are preferably written using Java™, which means that the programs only need to be written once, whereupon they will work on any device having a Java Virtual Machine running. The programs could be written using C++ or C# (C-sharp), whereupon they would have to be compiled for each device type (or more specifically, for each operating system type/processor type combination). As a further alternative the programs could be written using an interpreted language like Visual Basic™, but the GUI created therewith could only be run on devices running Microsoft™ operating systems, and would have to be compiled specifically for other operating systems.

If the programs were written using the C++programming language, the functionality of the interface 701 would be provided by a so-called virtual base class and pure virtual member functions (mentioned in the introduction).

CLAIMS

1. Apparatus for creating a Graphical User Interface for a device, comprising

a store arranged to store a plurality of processable procedures, each being associated with one or more device types;

receiving means arranged to receive a device independent Graphical User Interface specification and data identifying the device;

processing means arranged to process the received Graphical User Interface specification in accordance with the received identifying data, so as to identify, from the store, a plurality of procedures corresponding to the identified device; and

combining means arranged to combine the identified procedures into a program for processing by the device, thereby creating the Graphical User Interface.

2. Apparatus according to claim 1, wherein the device independent Graphical User Interface specification comprises a plurality of items arranged hierarchically with respect to one another.

3. Apparatus according to claim 1 or claim 2, wherein at least some of the plurality of procedures are written in accordance with object-oriented programming, so that the at least some procedures have, associated therewith, a class, which is associated with a device type.

4. Apparatus according to claim 3, further including means arranged to instantiate an object of the identified device type and to pass the instantiated object to the processing means, the processing means thereby having access to the procedures associated with the instantiated object.

5. Apparatus according to claim 4, wherein the processing means is arranged to receive a reference to procedures corresponding to items in the received Graphical User Interface specification.

6. Apparatus according to claim 2, further including a look-up table detailing mappings between said items and device-specific procedures, wherein for each of

said items in the Graphical User Interface specification, the processing means is arranged to select, from the look-up table, procedures that correspond to the said item.

7. Apparatus according to any one of the preceding claims, wherein the processable procedures include procedures for processing one or more events in accordance with user input received via the application program.

8. Apparatus according to claim 7, including means to select one or more of the said procedures for processing one or more events on the device in accordance with items in the Graphical User Interface specification.

9. Server apparatus for creating Graphical User Interface comprising
means arranged to receive a request for a Graphical User Interface from a device;
apparatus according to any one of the above claims; and
transmitting means arranged to transmit, to the requesting device, the program.

10. Server apparatus according to claim 9 when dependent on claim 8, wherein the transmitting means is further arranged to transmit, to the requesting device, the or each selected procedures.

11. A device comprising
apparatus according to any one of claims 1 to 8;
means arranged to receive the said program, and
means arranged to process the received program, thereby displaying the graphical user interface on the device.

12. A device according to claim 11 further including means arranged to receive user input via the received program and to process one or more events on the device in accordance with the received input.

13. A method of creating a Graphical User Interface for a device, comprising
receiving data identifying the device;

receiving a device independent Graphical User Interface specification;

accessing a store arranged to store a plurality of processable procedures, each being associated with one or more device types;

processing the received Graphical User Interface specification in accordance with the received device data so as to identify, from the store, one or more procedures corresponding to the received device data; and

combining the identified procedures into a program for processing by the device, thereby creating the Graphical User Interface.

14. A method according to claim 13, in which the Graphical User Interface specification comprises a plurality of display items arranged hierarchically with respect to one another, and the step of processing the received Graphical User Interface specification involves

reading in a first display item occurring at the top of the hierarchy;

selecting a procedure corresponding both to the device type and to the first display item;

recursively identifying items connected to the first display item; and

for each identified item, selecting a procedure corresponding both to the device type and to the identified display item.

15. Apparatus for creating a program for a device, comprising

a store arranged to store a plurality of processable procedures, each being associated with one or more device types;

receiving means arranged to receive, or to access, data identifying the device and a device independent specification;

processing means arranged to process the received specification in accordance with the received identifying data, thereby identifying, from the store, one or more procedures corresponding to the identified device;

combining means arranged to combine the identified procedures into a program for processing by the device.

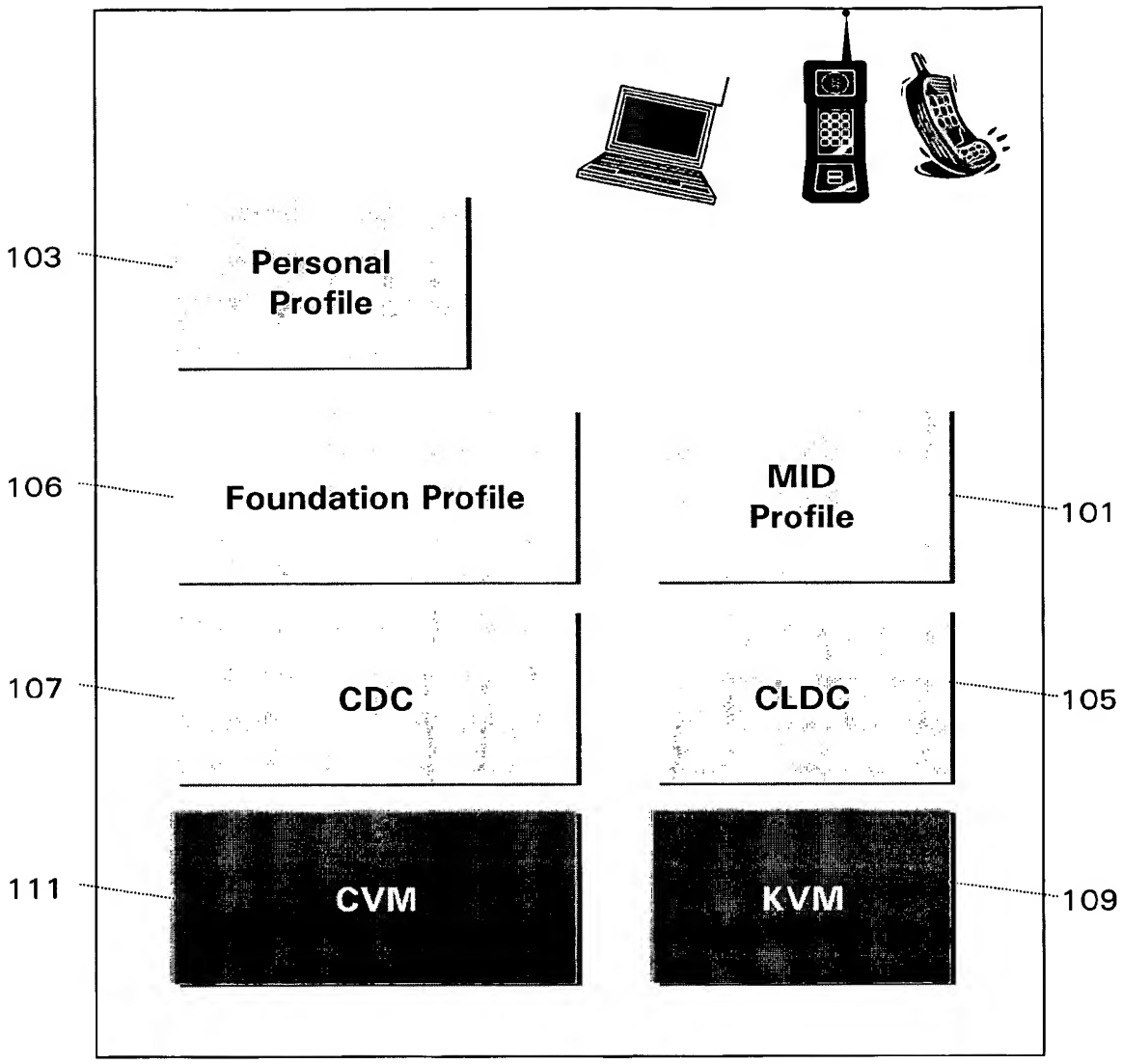
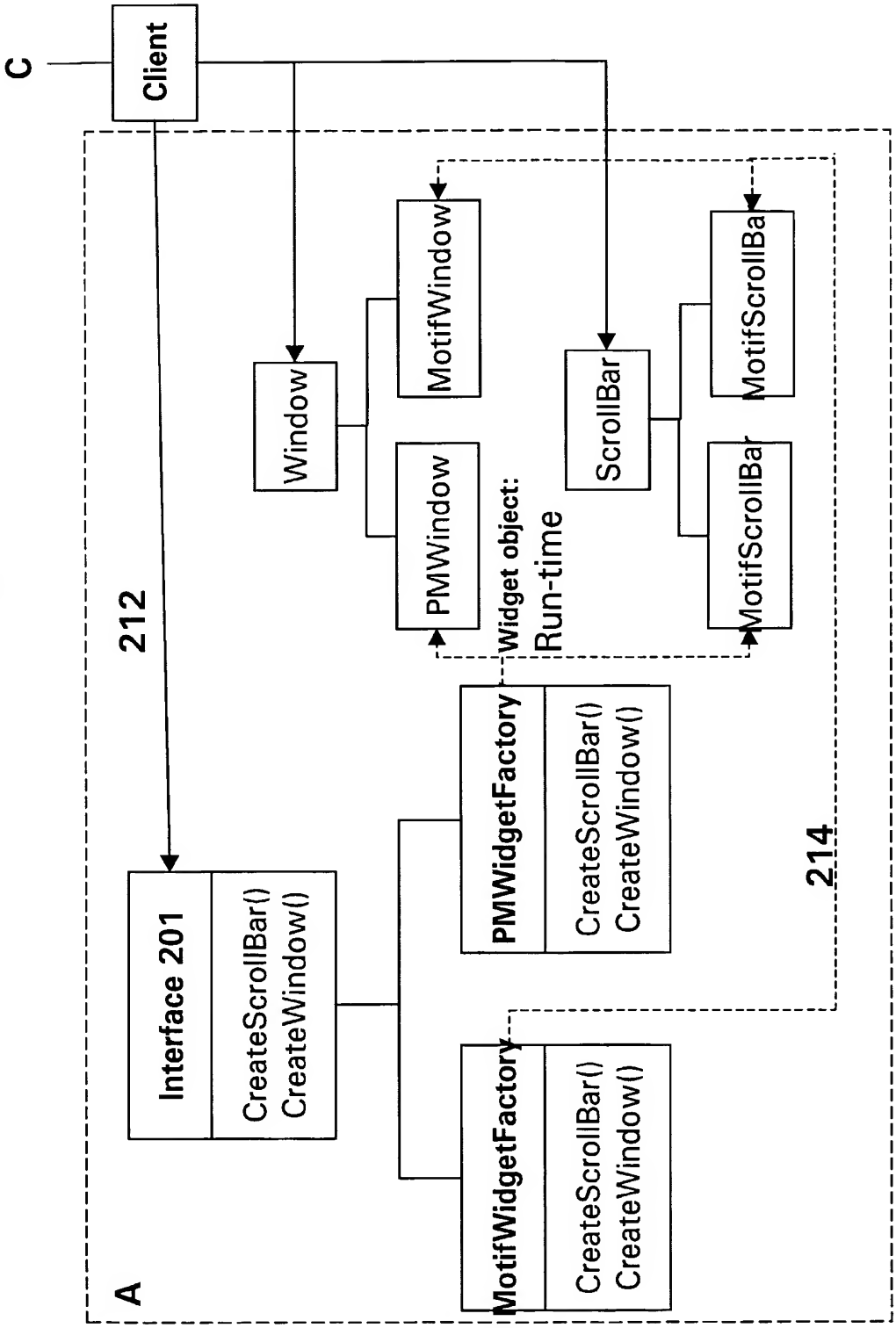

103
Personal Profile
106
Foundation Profile
MID Profile
101
107
CDC
CLDC
105
111
CVM
KVM
109

Fig 1

Fig 2


C
Client
Window
MotifWindow
PMWindow
ScrollBar
MotifScrollBa
MotifScrollBar
Widget object:
Run-time
212
214
Interface 201
CreateScrollBar()
CreateWindow()
PMWidgetFactory
CreateScrollBar()
CreateWindow()
MotifWidgetFactory
CreateScrollBar()
CreateWindow()
A

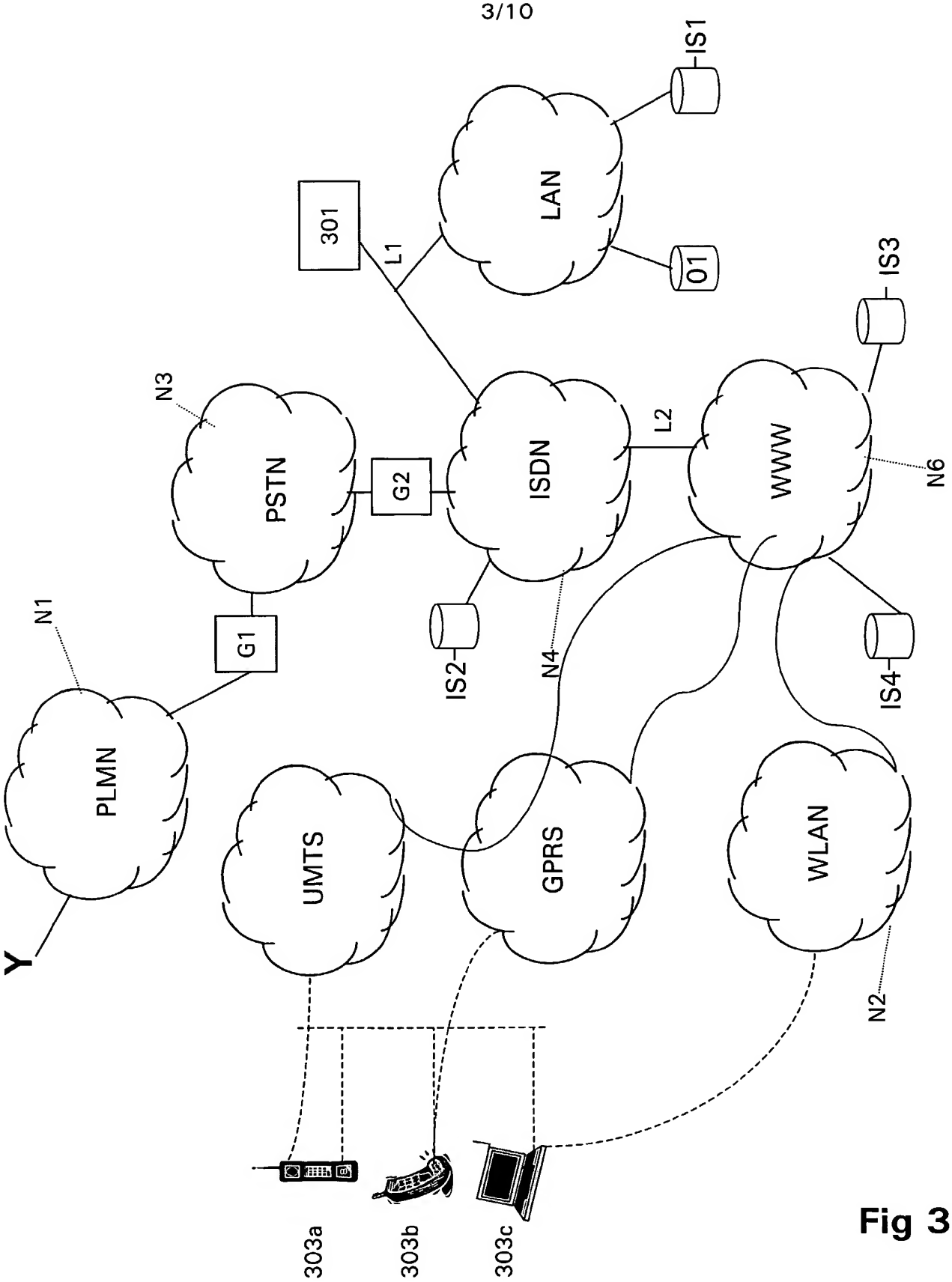

IS1
LAN
301
L1
01
IS3
N3
PSTN
G2
ISDN
L2
WWW
N6
N1
G1
IS2
N4
IS4
PLMN
UMTS
GPRS
WLAN
Y
N2
303a
303b
303c

Fig 3

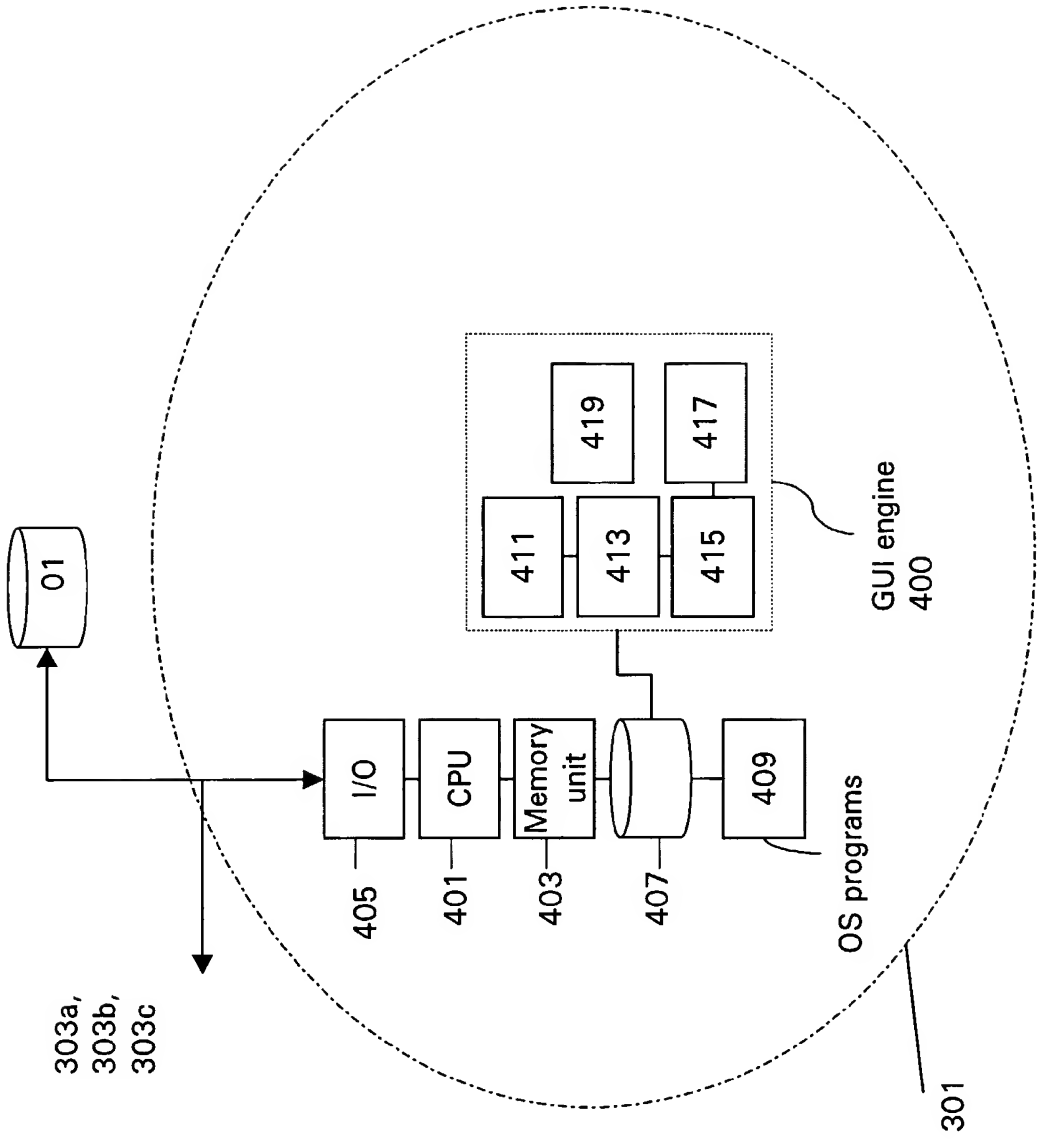

01
419
417
411
413
415
GUI engine
400
I/O
CPU
Memory unit
409
405
401
403
407
OS programs
303a,
303b,
303c
301

Fig 4

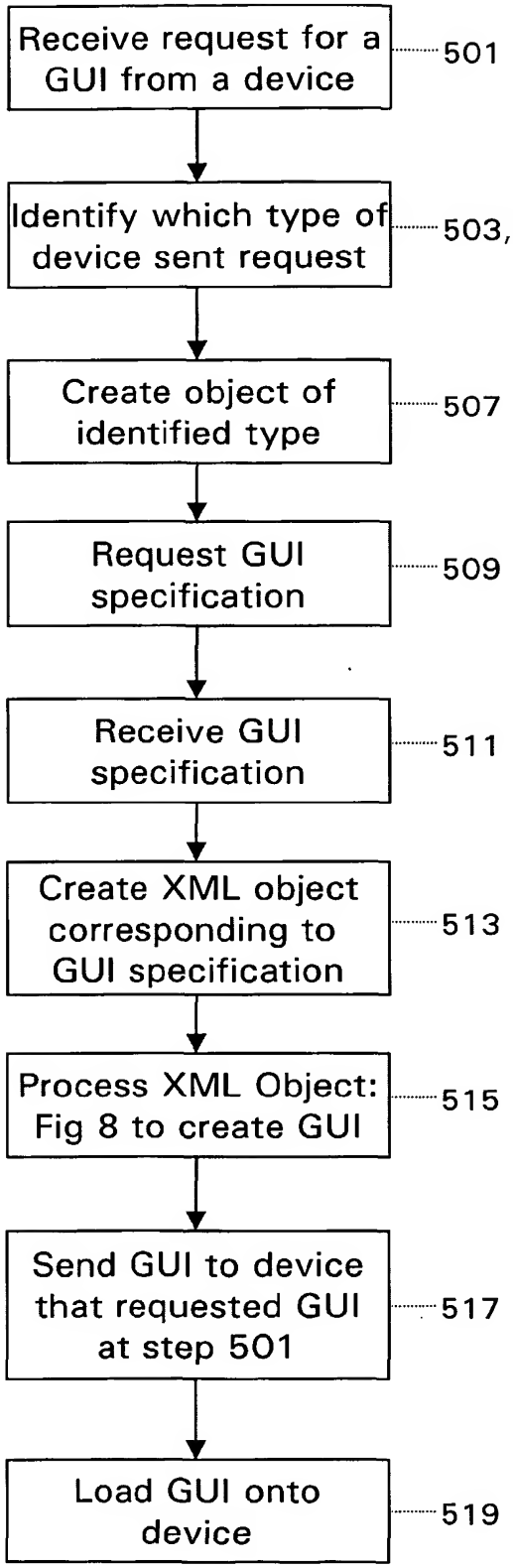

Receive request for a GUI from a device
501
Identify which type of device sent request
503,505
Create object of identified type
507
Request GUI specification
509
Receive GUI specification
511
Create XML object corresponding to GUI specification
513
Process XML Object: Fig 8 to create GUI
515
Send GUI to device that requested GUI at step 501
517
Load GUI onto device
519

Fig 5

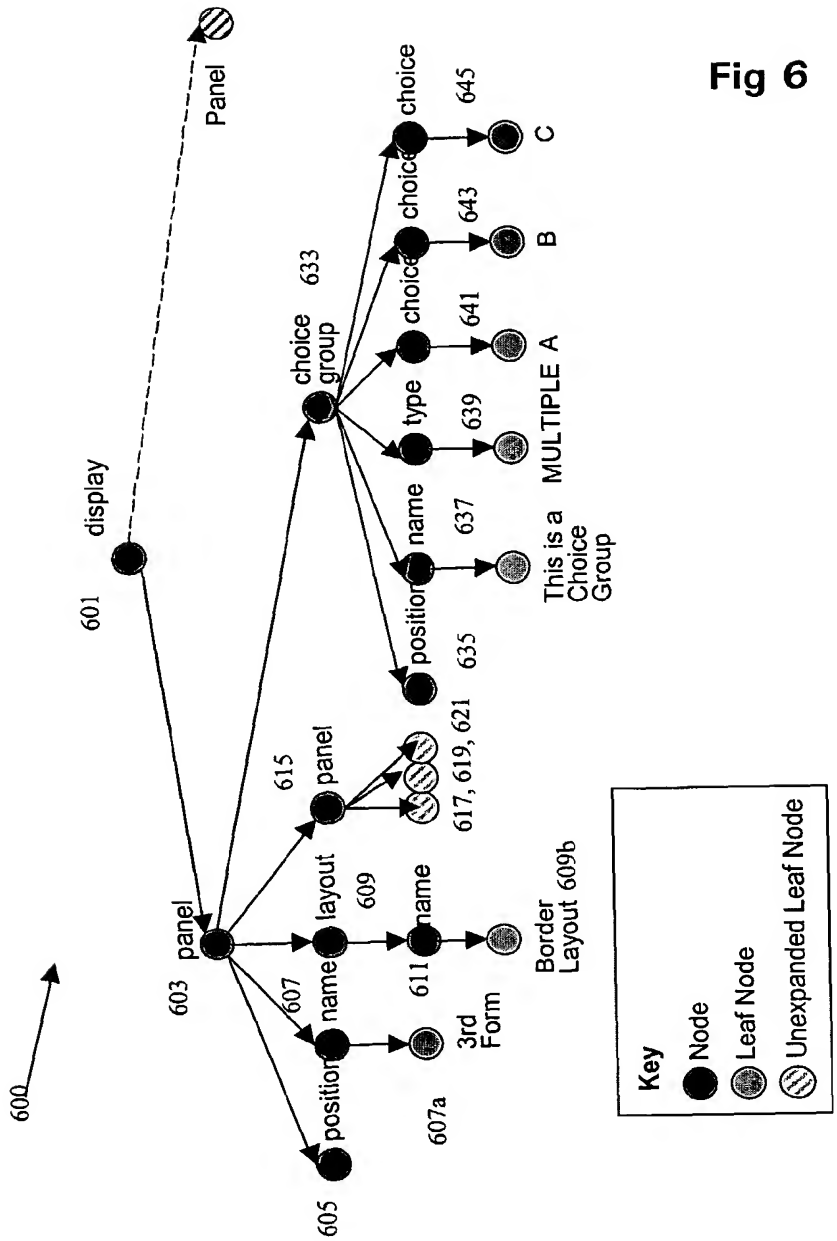
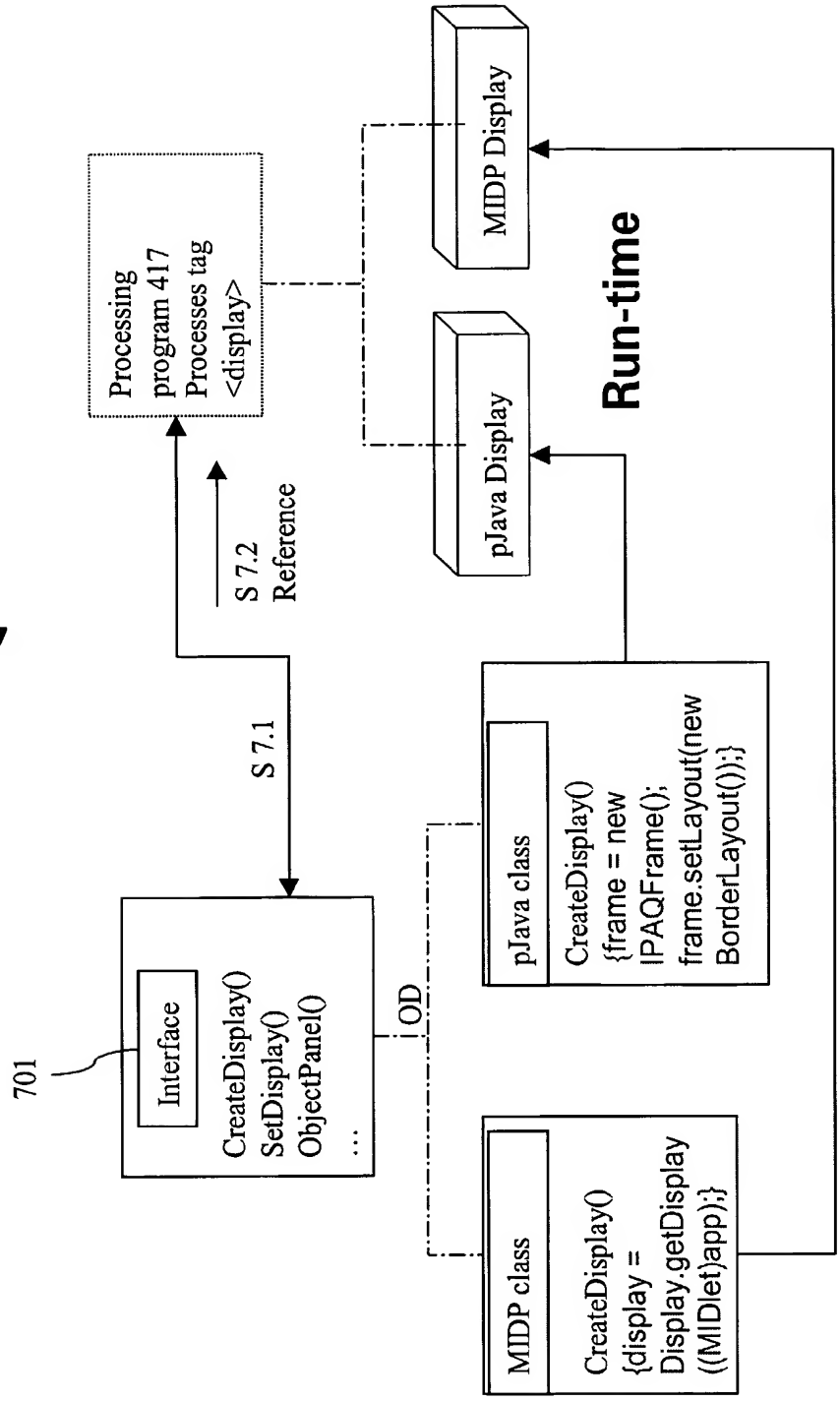
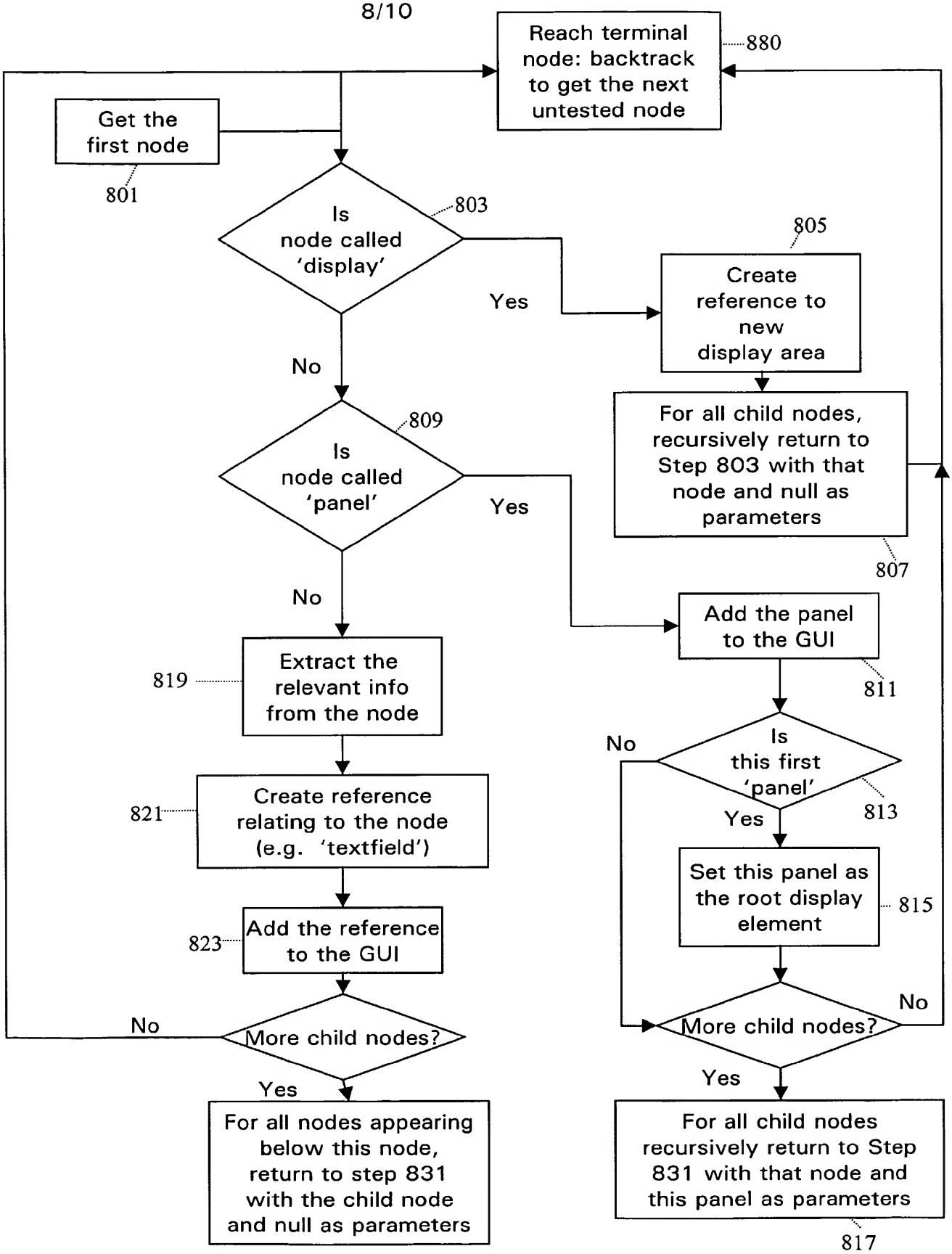
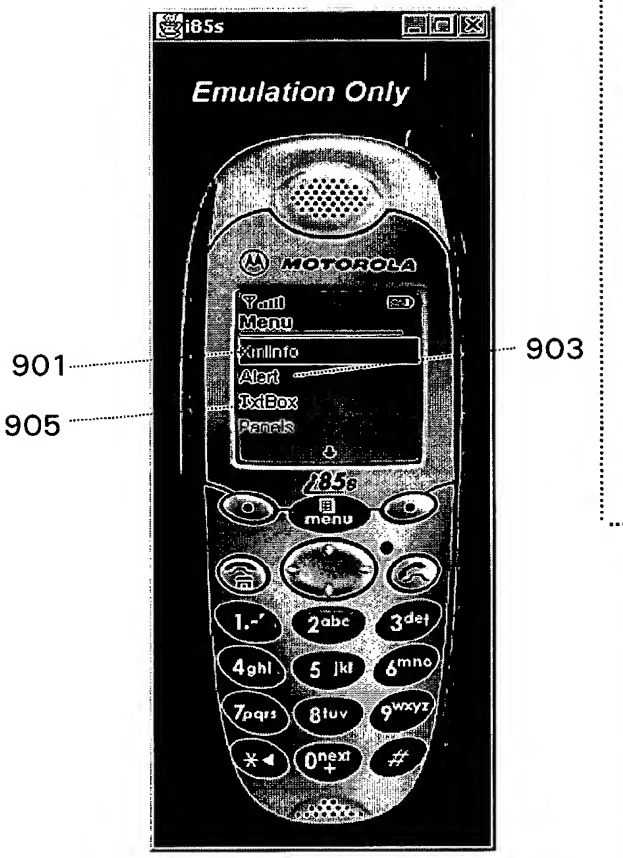

Panel
display
601
choice group
633
choice
645
C
choice
643
B
choice
641
A
type
639
MULTIPLE
name
637
This is a Choice Group
position
635
panel
615
617, 619, 621
panel
603
layout
609
name
611
Border Layout
609b
name
607
position
3rd Form
607a
605
600
Key
Node
Leaf Node
Unexpanded Leaf Node

Fig 6

Fig 7


Processing program 417 Processes tag <display>
MIDP Display
pJava Display
Run-time
S 7.2 Reference
S 7.1
701
Interface
CreateDisplay()
SetDisplay()
ObjectPanel()
...
OD
pJava class
CreateDisplay()
{frame = new
IPAQFrame();
frame.setLayout(new
BorderLayout());}
MIDP class
CreateDisplay()
{display =
Display.getDisplay
((MIDlet)app);}


Reach terminal node: backtrack to get the next untested node
880
Get the first node
801
Is node called 'display'
803
Yes
805
Create reference to new display area
No
For all child nodes, recursively return to Step 803 with that node and null as parameters
807
Is node called 'panel'
809
Yes
No
Add the panel to the GUI
811
Extract the relevant info from the node
819
No
Is this first 'panel'
813
Yes
Create reference relating to the node (e.g. 'textfield')
821
Set this panel as the root display element
815
Add the reference to the GUI
823
No
More child nodes?
No
Yes
Yes
For all nodes appearing below this node, return to step 831 with the child node and null as parameters
For all child nodes recursively return to Step 831 with that node and this panel as parameters
817


i85s
Emulation Only
MOTOROLA
Menu
XmlInfo
Alert
TxtBox
Panels
menu
901
903
905

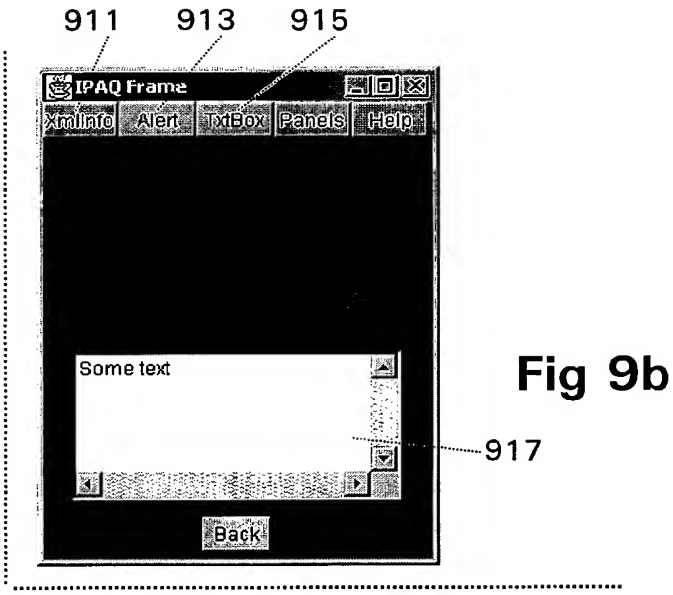

911
913
915
IPAQ Frame
XmlInfo
Alert
TxtBox
Panels
Help
Some text
Back
917

Fig 9b


i85s
Emulation Only
MOTOROLA
#hform
TEXT AREA
Some text
Back
AppMenu
menu
907

Fig 9a

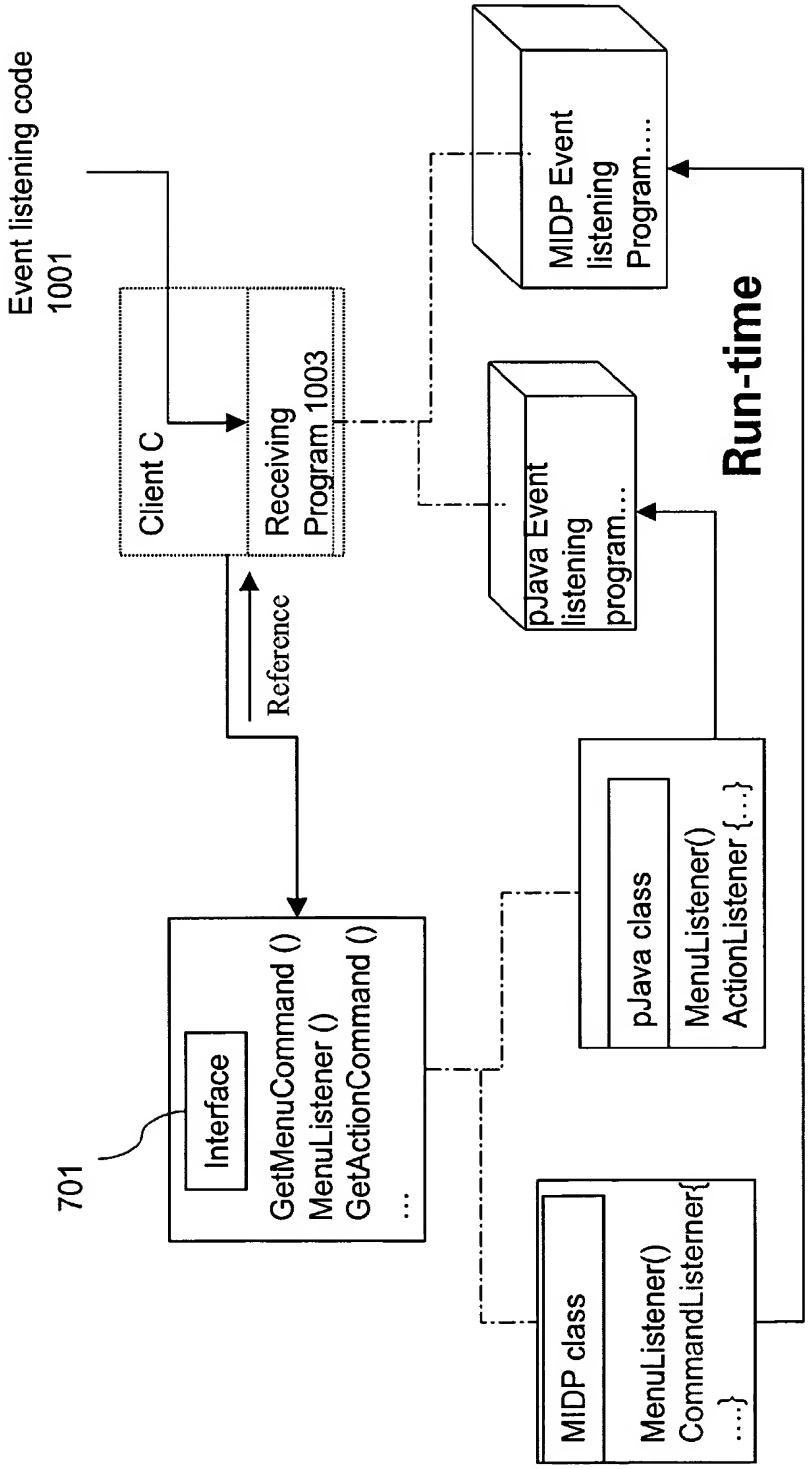

Event listening code
1001
Client C
Receiving
Program 1003
Reference
MIDP Event
listening
Program....
pJava Event
listening
program...
Run-time
701
Interface
GetMenuCommand ()
MenuListener ()
GetActionCommand ()
...
pJava class
MenuListener()
ActionListener {...}
MIDP class
MenuListener()
CommandListener{
....}

Fig 10